

# Algorithms for Replica Placement in High-Availability Storage

K. Alex Mills, R. Chandrasekaran, Neeraj Mittal

Department of Computer Science  
The University of Texas at Dallas, Richardson Texas, USA  
{k.alex.mills,chandra,neerajm}@utdallas.edu

**Abstract.** A new model of causal failure is presented and used to solve a novel replica placement problem in data centers. The model describes dependencies among system components as a directed graph. A replica placement is defined as a subset of vertices in such a graph. A criterion for optimizing replica placements is formalized and explained. In this work, the optimization goal is to avoid choosing placements in which a single failure event is likely to wipe out multiple replicas. Using this criterion, a fast algorithm is given for the scenario in which the dependency model is a tree. The main contribution of the paper is an  $O(n + \rho^2)$  dynamic programming algorithm for placing  $\rho$  replicas on a tree with  $n$  vertices. This algorithm exhibits the interesting property that only *two* subproblems need to be recursively considered at each stage. An  $O(n^2\rho)$  greedy algorithm is also briefly reported.

## 1 Introduction

With the surge towards the cloud, our websites, services and data are increasingly being hosted by third-party data centers. These data centers are often contractually obligated to ensure that data is rarely, if ever unavailable. One cause of unavailability is co-occurring component failures, which can result in outages that can affect millions of websites [13], and can cost millions of dollars in profits [11]. An extensive one-year study of availability in Google’s cloud storage infrastructure showed that such failures are relatively harmful. Their study emphasizes that “correlation among node failure dwarfs all other contributions to unavailability in our production environment” [4].

We believe that the correlation found among failure events arises due to dependencies among system components. Much effort has been made in the literature to produce quality statistical models of this correlation. But in using such models researchers do not make use of the fact that these dependencies can be explicitly modeled, since they are known to the system designers. In contrast, we propose a model wherein such dependencies are included, and demonstrate how an algorithm may make use of this information to optimize placement of data replicas within the data center.

To achieve high availability, data centers typically store multiple replicas of data to tolerate the potential failure of system components. This gives rise to a

*placement problem*, which, broadly speaking, involves determining which subset of nodes in the system should store a copy of a given file so as to maximize a given objective function (*e.g.*, reliability, communication cost, response time, or access time). While our focus is on replica placements, we note that our model could also be used to place replicas of other system entities which require high-availability, such as virtual machines and mission-critical tasks.

In this work, we present a new model for causal dependencies among failures, and a novel algorithm for optimal replica placement in our model. An example model is given as Fig. 1, in which three identical replicas of the same block of data are distributed on servers in a data center. Each server receives power from a surge protector which is located on each server rack. In Scenario I, each replica is located on nodes which share the same rack. In Scenario II, each replica is located on separate racks. As can be seen from the diagram of Scenario I (Fig. 1(a)), a failure in the power supply unit (PSU) on a single rack could result in a situation where every replica of a data block is completely unavailable, whereas in Scenario II, (Fig. 1(b)) three PSUs would need to fail in order to achieve the same result. In practice, Scenario I is avoided by ensuring that each replica is placed on nodes which lie on separate racks. This heuristic is already part of known best-practices. Our observation is that this simple heuristic can be suboptimal under certain conditions. For example, consider a failure in the aggregation switch which services multiple racks. Such a failure could impact the availability of every data replica stored on the rack. Moreover, this toy example only represents a small fraction of the number of events that could be modeled in a large data center.

While many approaches for replica placement have been proposed, our approach of modeling causal dependencies among failure events appears to be new. Other work on reliability in storage area networks has focused on objectives such as mean time to data loss [3, 7]. These exemplify an approach towards correlated failure which we term “measure-and-conquer”. In measure-and-conquer approaches, a measured degree of correlation is given as a parameter to the model. In contrast, we model explicit causal relations among failure events which we believe give rise to the correlation seen in practice. In [7] the authors consider high-availability replica placement, but are primarily focused on modeling the effects of repair time. Later work [3] begins to take into account information

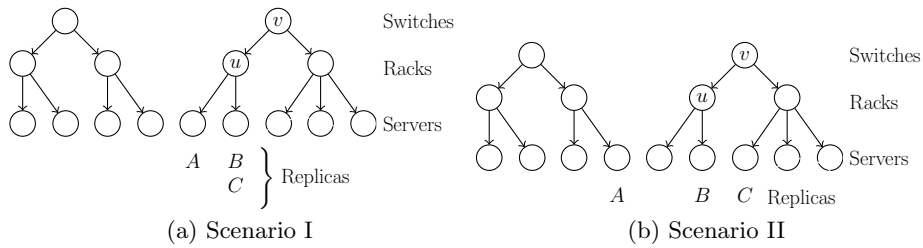


Fig. 1: Two scenarios represented by directed trees.

concerning the network topology, which is a step towards our approach. Similar measure-and-conquer approaches are taken in [1, 4, 8, 14]. More recently, Pezoa and Hayat [10] have presented a model in which spatially correlated failures are explicitly modeled. However, they consider the problem of task allocation, whereas we are focused on replica placement. In the databases community, work on replica placement primarily focuses on finding optimal placements in storage-area networks with regard to a particular distributed access model or mutual exclusion protocol [5, 12, 15]. In general, much of the work from this community focuses on specialized communication networks and minimizing communication costs — system models and goals which are substantially different from our own.

Recently, there has been a surge of interest in computer science concerning cascading failure in networks [2, 6, 9, 16]. While our model is most closely related to this work, the existing literature is primarily concerned with applications involving large graphs intended to capture the structure of the world-wide web, or power grids. The essence of all these models is captured in the *threshold cascade model* [2]. This model consists of a directed graph in which each node  $v$  is associated with a threshold,  $\ell(v) \in \mathbb{N}^+$ . A node  $v$  experiences a cascading failure if at least  $\ell(v)$  of its incoming neighbors have failed. This model generalizes our own, wherein we pessimistically assume that  $\ell(v) = 1$  for all nodes  $v$ . Current work in this area is focused on network design [2], exploring new models [6, 9], and developing techniques for adversarial analysis [16]. To our knowledge, no one has yet considered the problem of replica placement in such models.

## 2 Model

We model dependencies among failure events as a directed graph, where nodes represent failure events, and a directed edge from  $u$  to  $v$  indicates that the occurrence of failure event  $u$  could trigger the occurrence of failure event  $v$ . We refer to this graph as the *failure model*.

Given such a graph as input, we consider the problem of selecting nodes on which to store data replicas. Roughly, we define a *placement problem* as the problem of selecting a subset of these vertices, hereafter referred to as a *placement*, from the failure model so as to satisfy some safety criterion. In our application, only those vertices which represent storage servers are candidates to be part of a placement. We refer to such vertices as *placement candidates*. Note that the graph also contains vertices representing other types of failure events, which may correspond to real-world hardware unsuitable for storage (such as a ToR switch), or even to abstract events which have no associated physical component. In most applications, the set of placement candidates forms a subset of the set of vertices.

More formally, let  $E$  denote the set of failure events, and  $C$  denote the set of placement candidates. We are interested in finding a *placement* of size  $\rho$ , which is defined to be a set  $P \subseteq C$ , with  $|P| = \rho$ . Throughout this paper we will use  $P$  to denote a placement, and  $\rho$  to denote its size. We consistently use  $C$  to denote the set of placement candidates, and  $E$  to denote the set of failure events.

Let  $G = (V, A)$  be a directed graph with vertices in  $V$  and edges in  $A$ . The vertices represent both events in  $E$  and candidates in  $C$ , so let  $V = E \cup C$ . A directed edge between events  $e_1$  and  $e_2$  indicates that the occurrence of failure event  $e_1$  can trigger the occurrence of failure event  $e_2$ . A directed edge between event  $e$  and candidate  $c$  indicates that the occurrence of event  $e$  could compromise candidate  $c$ . We will assume failure to act transitively. That is, if a failure event occurs, all failure events reachable from it in  $G$  also occur. This a pessimistic assumption which leads to a conservative interpretation of failure.

We now define the notions of *failure number* and *failure aggregate*.

**Definition 1.** Let  $e \in E$ . The failure number of event  $e$ , denoted  $f(e, P)$ , for a given placement  $P$ , is defined as the number of candidates in  $P$  whose correct operation could be compromised by occurrence of event  $e$ . In particular,

$$f(e, P) = |\{p \in P \mid p \text{ is reachable from } e \text{ in } G\}|.$$

As an example, node  $u$  in Fig. 1 has failure number 3 in Scenario I, and failure number 1 in Scenario II. The following property is an easy consequence of the above definition. A formal proof can be found in the appendix.

*Property 1.* For any placement  $P$  of replicas in tree  $T$ , if node  $i$  has descendant  $j$ , then  $f(j, P) \leq f(i, P)$ .

The failure number captures a conservative criterion for a safe placement. Intuitively, we consider the worst case scenario, in which every candidate which *could* fail due to an occurring event *does* fail. Our goal is to find a placement which does not induce large failure numbers in any event. To aggregate this idea across all events, we define *failure aggregate*, a measure that accounts for the failure number of every event in the model.

**Definition 2.** The failure aggregate of a placement  $P$  is a vector in  $\mathbb{N}^{p+1}$ , denoted  $\mathbf{f}(P)$ , where  $\mathbf{f}(P) := \langle p_\rho, \dots, p_1, p_0 \rangle$ , and each  $p_i := |\{e \in E \mid f(e, P) = i\}|$ .

In Fig. 1, node  $v$  has failure aggregate  $\langle 2, 0, 0, 1 \rangle$  in Scenario I and failure aggregate  $\langle 1, 0, 2, 0 \rangle$  in Scenario II. Failure aggregate is also computed in Fig. 6.

In all of the problems considered in this paper, we are interested in optimizing  $\mathbf{f}(P)$ . When optimizing a vector quantity, we must choose a meaningful way to totally order the vectors. In the context of our problem, we find that ordering the vectors with regard to the *lexicographic order* is both meaningful and convenient. The lexicographic order  $\leq_L$  between  $\mathbf{f}(P) = \langle p_\rho, \dots, p_1, p_0 \rangle$  and  $\mathbf{f}(P') = \langle p'_\rho, \dots, p'_1, p'_0 \rangle$  is defined via the following formula:

$$\mathbf{f}(P) \leq_L \mathbf{f}(P') \iff \exists m > 0, \forall i > m [p_i = p'_i \wedge p_m \leq p'_m].$$

To see why this is desirable, consider a placement  $P$  which lexicominimizes  $\mathbf{f}(P)$  among all possible placements. Such a placement is guaranteed to minimize  $p_\rho$ , i.e. the number of nodes which compromise *all* of the entities in our placement. Further, among all solutions minimizing  $p_\rho$ ,  $P$  also minimizes  $p_{\rho-1}$ , the

number of nodes compromising *all but one* of the entities in  $P$ , and so on for  $p_{\rho-2}, p_{\rho-3}, \dots, p_0$ . Clearly, the lexicographic order nicely prioritizes minimizing the entries of the vector in an appealing manner.

Throughout the paper, any time a vector quantity is maximized or minimized, we are referring to the maximum or minimum value in the lexicographic order. We will also use  $\mathbf{f}(P)$  to denote the failure aggregate, and  $p_i$  to refer to the  $i^{th}$  component of  $\mathbf{f}(P)$ , where  $P$  can be inferred from context.

In the most general case, we could consider the following problem.

*Problem 1.* Given graph  $G = (V, A)$  with  $V = C \cup E$ , and positive integer  $\rho$  with  $\rho < |C|$ , find a placement  $P \subseteq C$  with  $|P| = \rho$  such that  $\mathbf{f}(P)$  is lexicominimum.

Problem 1 is NP-hard to solve, even in the case where  $G$  is a bipartite graph. In particular, a reduction to independent set can be shown. However, the problem is tractable for special classes of graphs, one of which is the case wherein the graph forms a directed, rooted tree with leaf set  $L$  and  $C = L$ . Our main contribution in this paper is a fast algorithm for solving Problem 1 in such a case. We briefly mention a greedy algorithm which solves the problem on  $O(n^2\rho)$  time. However, since  $n \gg \rho$  in practice our result of an  $O(n + \rho^2)$  algorithm is much preferred.

## 2.1 An $O(n^2\rho)$ Greedy Algorithm

The greedy solution to this problem forms a partial placement  $P'$ , to which new replicas are added one at a time, until  $\rho$  replicas have been placed overall.  $P'$  starts out empty, and at each step, the leaf  $u$  which lexicominimizes  $\mathbf{f}(P' \cup \{u\})$  is added to  $P'$ . This greedy algorithm correctly computes an optimal placement, however its running time is  $O(n^2\rho)$  for a tree of unbounded degree. This running time comes about since each iteration requires visiting  $O(|L|)$  leaves for inclusion. For each leaf  $q$  which is checked, every node on a path from  $q$  to the root must have its failure number computed. Both the length of a leaf-root path and the number of leaves can be bounded by  $O(n)$  in the worst case, yielding the result.

That the greedy algorithm works correctly is not immediately obvious. It can be shown via an exchange argument that each partial placement found by the greedy algorithm is a subset of some optimal placement. This is the content of Theorem 1 below.

To establish the correctness of the greedy algorithm, we first introduce some notation. For a placement  $P$  and  $S \subseteq V$ , let  $\mathbf{f}(S, P) = \langle g_\rho, g_{\rho-1}, \dots, g_1, g_0 \rangle$  where  $g_i := |\{x \in S \mid f(x, P) = i\}|$ . Intuitively,  $\mathbf{f}(S, P)$  gives the failure aggregate for all nodes in set  $S \subseteq V$ . We first establish the truth of two technical lemmas before stating and proving Theorem 1.

**Lemma 1.** *Let  $r$  be the root of a failure model given by a tree. Given  $P \subseteq C$ ,  $a, b \in C - P$ . If  $f(r \rightsquigarrow a, P) <_L f(r \rightsquigarrow b, P)$  then  $f(P \cup \{a\}) <_L f(P \cup \{b\})$ .*

*Proof.* Suppose  $f(r \rightsquigarrow a, P) <_L f(r \rightsquigarrow b, P)$ . Let nodes on the paths from  $r$  to  $a$  and from  $r$  to  $b$  be labeled as follows:

$$r \rightarrow a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow a$$

$$r \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_m \rightarrow b$$

We proceed in two cases.

In the first case, there is some  $1 \leq i \leq \min(m, n)$  for which  $f(a_i, P) < f(b_i, P)$ . Let  $i$  be the minimum such index, and let  $f(b_i, P) = k$ . Clearly,  $f(P \cup \{a\})_k < f(P \cup \{b\})_k$ , since  $P \cup \{b\}$  counts  $b_i$  as having survival number  $k$  and  $P \cup \{a\}$  does not. Moreover, since  $f(a_\ell, P) = f(b_\ell, P)$  for all  $\ell < i$ , we have that for all  $j > k$ ,  $f(P \cup \{a\})_j = f(P \cup \{b\})_j$  by Property 1.

In the second case,  $f(a_i, P) \geq f(b_i, P)$  for all  $1 \leq i \leq \min(m, n)$ . In this case, if  $f(a_i, P) > f(b_i, P)$  for some  $i$ , the only way we could have  $f(r \rightsquigarrow a, P) <_L f(r \rightsquigarrow b, P)$  is if there is some  $j > i$  with  $f(a_j, P) < f(b_j, P)$ , but this is a contradiction. Therefore,  $f(a_i, P) = f(b_i, P)$  for all  $1 \leq i \leq \min(m, n)$ . So, we must also have  $n \leq m$ , since if  $n > m$ , we would have  $f(r \rightsquigarrow a, P) >_L f(r \rightsquigarrow b, P)$ . Moreover, since  $f(r \rightsquigarrow a, P) <_L f(r \rightsquigarrow b, P)$ , we must have that  $n < m$ , for if  $n = m$ , we would have  $f(r \rightsquigarrow a, P) = f(r \rightsquigarrow b, P)$ , a contradiction. We have just shown the existence of some node  $b_{n+1}$ , for which we must have that  $f(b_{n+1}, P) \leq f(a_n, P)$ . Notice that the path  $r \rightsquigarrow a$  does not have an  $(n+1)^{st}$  node, so it's clear that if  $f(b_{n+1}, P) = k$ , then  $f(P \cup \{a\})_k < f(P \cup \{b\})_k$ . Finally, since  $n < m$ , we have by Property 1, that  $f(a_i, P) \leq f(a_n, P) \leq k$  for all  $1 \leq i \leq n$ . By an additional application of Property 1 it's easy to see that for all  $j > k$ , we have  $f(P \cup \{a\})_j = f(P \cup \{b\})_j$ .  $\square$

From Lemma 1, we obtain the following result as an easy Corollary.

**Corollary 1.** *Let  $r$  be the root of a failure model given by a tree. Given  $P \subseteq C$ ,  $a, b \in C - P$ . Then  $f(r \rightsquigarrow a, P) \leq_L f(r \rightsquigarrow b, P)$  if and only if  $f(P \cup \{a\}) \leq_L f(P \cup \{b\})$ .*

*Proof.* Suppose  $f(r \rightsquigarrow a, P) \leq_L f(r \rightsquigarrow b, P)$ . If  $f(r \rightsquigarrow a, P) = f(r \rightsquigarrow b, P)$ , then since the only nodes which change failure number when considering placements  $P$  and  $P \cup \{a\}$  are those on the paths  $r \rightsquigarrow a$ , and each of these nodes' failure numbers increase by 1, we must have that  $f(P \cup \{a\}) = f(P \cup \{b\})$ , since the sequence of failure numbers in  $r \rightsquigarrow a$  and  $r \rightsquigarrow b$  are the same. If  $f(r \rightsquigarrow a, P) <_L f(r \rightsquigarrow b, P)$  then by Lemma 1 the Corollary is proven.

If instead  $f(P \cup \{a\}) \leq_L f(P \cup \{b\})$ , and yet  $f(r \rightsquigarrow a, P) >_L f(r \rightsquigarrow b, P)$ , then by Lemma 1 we obtain that  $f(P \cup \{a\}) >_L f(P \cup \{b\})$ , a contradiction.  $\square$

Given a node  $u$  in a tree, let  $L(u)$  be the set of all leaves which are descendants of  $u$ .

**Lemma 2.** *Given  $P \subseteq C$ ,  $a, b \in C$ . Let  $c$  be the least common ancestor of  $a$  and  $b$ , and let  $d$  be the child of  $c$  on the path from  $c$  to  $a$ . If  $f(r \rightsquigarrow a, P) \leq_L f(r \rightsquigarrow b, P)$  and  $X \subset C - \{a, b\}$  for which  $L(d) \cap X = \emptyset$ , and  $a, b \notin X$  then*

$$f(P \cup X \cup \{a\}) \leq_L f(P \cup X \cup \{b\}).$$

*Proof.* We have that  $f(r \rightsquigarrow a, P) \leq_L f(r \rightsquigarrow b, P)$ . Consider  $f(r \rightsquigarrow a, P \cup X)$  and  $f(r \rightsquigarrow b, P \cup X)$ . We wish to show that  $f(r \rightsquigarrow a, P \cup X) \leq_L f(r \rightsquigarrow b, P \cup X)$ .

Since  $c$  is the least common ancestor of  $a$  and  $b$ , it is clear that nodes on  $r \rightsquigarrow c$  have equivalent failure numbers in both cases. Therefore it suffices to show that  $f(c \rightsquigarrow a, P \cup X) \leq_L f(c \rightsquigarrow b, P \cup X)$ .

Note that since  $d \cap L(X) = \emptyset$ , we have that  $f(c \rightsquigarrow a, P \cup X) = f(c \rightsquigarrow a, P)$ . Moreover, since the addition of nodes in  $X$  cannot cause failure numbers on the path  $c \rightsquigarrow b$  to decrease, we must have that  $f(c \rightsquigarrow b, P) \leq_L f(c \rightsquigarrow b, P \cup X)$ . Altogether, we have that

$$f(c \rightsquigarrow a, P \cup X) = f(c \rightsquigarrow a, P) \leq_L f(c \rightsquigarrow b, P) \leq_L f(c \rightsquigarrow b, P \cup X).$$

By applying Corollary 1, we obtain that  $f(P \cup X \cup \{a\}) \leq_L f(P \cup X \cup \{b\})$ .  $\square$

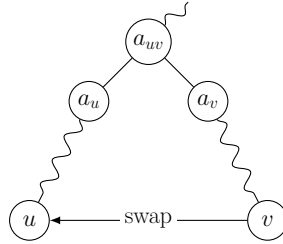


Fig. 2: Named nodes used in Theorem 1. The arrow labeled “swap” illustrates the leaf nodes between which replicas are moved, and is not an edge of the graph.

**Theorem 1.** *Let  $P_i$  be the partial placement from step  $i$  of the greedy algorithm. Then there exists an optimal placement  $P^*$ , with  $|P^*| = \rho$  such that  $P_i \subseteq P^*$ .*

*Proof.* The proof proceeds by induction on  $i$ .  $P_0 = \emptyset$  is clearly a subset of any optimal solution. Given  $P_i \subseteq P^*$  for some optimal solution  $P^*$ , we must show that there is an optimal solution  $Q^*$  for which  $P_{i+1} \subseteq Q^*$ . Clearly, if  $P_{i+1} \subseteq P^*$ , then we are done, since  $P^*$  is optimal. In the case where  $P_{i+1} \not\subseteq P^*$  we must exhibit some optimal solution  $Q^*$  for which  $P_{i+1} \subseteq Q^*$ . Let  $u$  be the leaf which was added to  $P_i$  to form  $P_{i+1}$ . Let  $v$  be the leaf in  $P^* - P_{i+1}$  which has the greatest-depth least common ancestor with  $u$ , where the depth of a node is given by its distance from the root (see Fig. 2). We set  $Q^* = (P^* - \{v\}) \cup \{u\}$ , and claim that  $\mathbf{f}(Q^*) \leq_L \mathbf{f}(P^*)$ . Since  $\mathbf{f}(P^*)$  is optimal, and  $P_{i+1} \subseteq Q^*$  this will complete our proof.

Clearly,  $f(a \rightsquigarrow u, P_i) \leq_L f(a \rightsquigarrow v, P_i)$ , since otherwise  $f(r \rightsquigarrow u, P_i) >_L f(r \rightsquigarrow v, P_i)$ , implying that  $f(P_i \cup \{u\}) >_L f(P_i \cup \{v\})$ , contradicting our use of a greedy algorithm.

Note that  $u, v \notin (P^* - P_i - \{v\})$ . Moreover, by choice of  $v$ , we have that  $L(a) \cap (P^* - P_i - \{v\}) = \emptyset$ , since the only nodes from  $P^*$  in  $L(a)$  must also be in  $P_i$ . To complete the proof, we apply Lemma 2, setting  $X = P^* - P_i - \{v\}$ . This choice of  $X$  is made so as to yield the following equalities.

$$Q^* = (P^* - \{v\}) \cup \{u\} = P_i \cup (P^* - P_i - \{v\}) \cup \{u\},$$

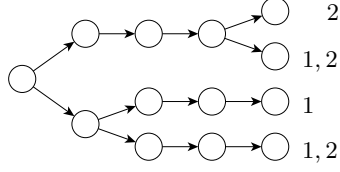


Fig. 3: Round-robin placement cannot guarantee optimality

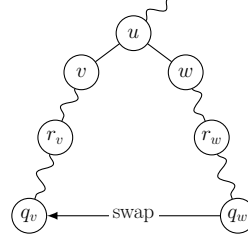


Fig. 4: Nodes used in Theorem 2.

$$P^* = P_i \cup (P^* - P_i - \{v\}) \cup \{v\}.$$

By Lemma 2, we obtain inequality in the following formula,

$$f(Q^*) = f(P_i \cup (P^* - P_i - \{v\}) \cup \{u\}) \leq_L f(P_i \cup (P^* - P_i - \{v\}) \cup \{v\}) = f(P^*).$$

Thereby completing the proof.  $\square$

### 3 Balanced Placements

Consider a round-robin placement in which the set of replicas placed at each node is distributed among its children, one replica per child, until all replicas have been placed. This process is then continued recursively at the children. Throughout the process, no child is given more replicas than its subtree has leaf nodes. This method has intuitive appeal, but it does not compute an optimal placement exactly as can be seen from Fig. 3. Let placements  $P_1$  and  $P_2$  consist of the nodes labeled by 1 and 2 in Fig. 3 respectively. Note that both outcomes are round-robin placements. A quick computation reveals that  $\mathbf{f}(P_1) = \langle 1, 1, 7, 0 \rangle \neq \langle 1, 3, 3, 2 \rangle = \mathbf{f}(P_2)$ . Since the placements have different failure aggregates, round-robin placement alone cannot guarantee optimality.

Key to our algorithm is the observation that any placement which lexicominimizes  $\mathbf{f}(P)$  must be *balanced*. If we imagine each child  $c_i$  of  $u$  as a bin of capacity  $\ell_i$ , balanced nodes are those in which all unfilled children are approximately “level”, and no child is filled while children of smaller capacity remain unfilled. These ideas are formalized in the following definitions.

**Definition 3.** Let node  $u$  have children indexed  $1, \dots, k$ , and let the subtree rooted at the  $i^{\text{th}}$  child of node  $u$  have  $\ell_i$  leaves, and  $r_i$  replicas placed on it in placement  $P$ . A node for which  $\ell_i - r_i = 0$  is said to be filled. A node for which  $\ell_i - r_i > 0$  is said to be unfilled.

**Definition 4.** Node  $u$  is said to be balanced in placement  $P$  iff:

$$\ell_i - r_i > 0 \implies \forall j \in \{1, \dots, k\} (r_i \geq r_j - 1).$$

Placement  $P$  is said to be balanced if all nodes  $v \in V$  are balanced.



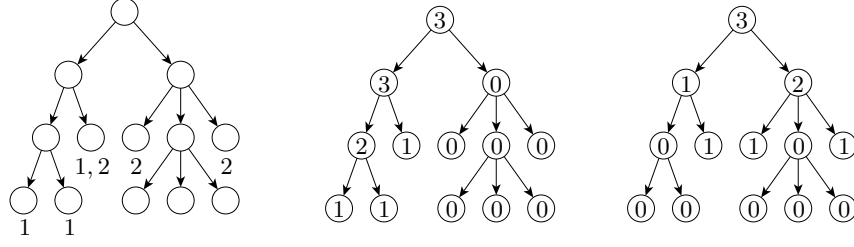


Fig. 5: Placements  $P_1, P_2$  Fig. 6: Failure numbers for  $P_1$  (right) and  $P_2$  (left).

To motivate a proof that lexico-minimum placements must be balanced, consider Fig. 5 in which  $P_1$  and  $P_2$  are sets containing leaf nodes labeled 1 and 2 respectively. Fig. 6 presents two copies of the same tree, but with failure numbers labeled according to  $P_1$  and  $P_2$ . Upon computing  $f(P_1)$  and  $f(P_2)$ , we find that  $f(P_1) = \langle 2, 1, 3, 7 \rangle \geq_L \langle 1, 1, 4, 7 \rangle = f(P_2)$ . Note that for placement  $P_1$ , the root of the tree is unbalanced, therefore  $P_1$  is unbalanced. Note also, that  $P_2$  is balanced, since each of its nodes are balanced. We invite the reader to verify that  $P_2$  is an optimal solution for this tree.

Our main result is that it is *necessary* for an optimal placement to be balanced. However, the balanced property alone is not sufficient to guarantee optimality. To see this, consider the two placements in Fig. 3. By definition, both placements are balanced, yet they have different failure aggregates. Therefore, balancing alone is insufficient to guarantee optimality. Despite this, we can use Theorem 2 to justify discarding unbalanced solutions as suboptimal. We exploit this property of optimal placements in our algorithm.

**Theorem 2.** *Any placement  $P$  in which  $\mathbf{f}(P)$  is lexicominimum among all placements for a given tree must be balanced.*

*Proof.* Suppose  $P$  is not balanced, yet  $\mathbf{f}(P)$  is lexicominimum among all placements  $P$ . We proceed to a contradiction, as follows.

Let  $u$  be an unbalanced node in  $T$ . Let  $v$  be an unfilled child of  $u$ , and let  $w$  be a child of  $u$  with at least one replica such that  $r_v < r_w - 1$ . Since  $v$  is unfilled, we can take one of the replicas placed on  $w$  and place it on  $v$ . Let  $q_w$  be the leaf node from which this replica is taken, and let  $q_v$  be the leaf node on which this replica is placed (see Fig. 4). Let  $P^* := (P - \{q_w\}) \cup \{q_v\}$ . We aim to show that  $P^*$  is more optimal than  $P$ , contradicting  $P$  as a lexicominimum.

Let  $\mathbf{f}(P) := \langle p_\rho, \dots, p_0 \rangle$ , and  $\mathbf{f}(P^*) := \langle p_\rho^*, \dots, p_0^* \rangle$ . For convenience, we let  $f(w, P) = m$ . To show that  $\mathbf{f}(P^*) <_L \mathbf{f}(P)$ , we aim to prove that  $p_m^* < p_m$ , and that for any  $k$  with  $\rho \geq k > m$ , that  $p_k^* = p_k$ . We will concentrate on proving the former, and afterwards show that the latter follows easily.

To prove  $p_m^* < p_m$ , observe that as a result of the swap, some nodes change failure number. These nodes all lie on the paths  $v \rightsquigarrow q_v$  and  $w \rightsquigarrow q_w$ . Let  $S^-$  (resp.  $S^+$ ) be the set of nodes whose failure numbers change to  $m$  (resp. change

from  $m$ ), as a result of the swap. Formally, we define

$$S^- := \{x \in V \mid f(x, P) = m, f(x, P^*) \neq m\},$$

$$S^+ := \{x \in V \mid f(x, P) \neq m, f(x, P^*) = m\}.$$

By definition,  $p_m^* = p_m - |S^-| + |S^+|$ . We claim that  $|S^-| \geq 1$  and  $|S^+| = 0$ , which yields  $p_m^* < p_m$ . To show  $|S^-| \geq 1$ , note that  $f(w, P) = m$  by definition, and after the swap, the failure number of  $w$  changes. Therefore,  $|S^-| \geq 1$ .

To show  $|S^+| = 0$ , we must prove that no node whose failure number is affected by the swap has failure number  $m$  after the swap has occurred. We choose to show a stronger result, that all such node's failure number must be strictly less than  $m$ . Let  $s_v$  be an arbitrary node on the path  $v \rightsquigarrow q_v$ , and consider the failure number of  $s_v$ . As a result of the swap, one more replica is counted as failed in each node on this path, therefore  $f(s_v, P^*) = f(s_v, P) + 1$ . Likewise, let  $s_w$  be an arbitrary node on path  $w \rightsquigarrow q_w$ . One less replica is counted as failed in each node on this path, so  $f(s_w, P^*) = f(s_w, P) - 1$ . We will show that  $f(s_w, P^*) < m$ , and  $f(s_v, P^*) < m$ .

First, note that for any  $s_w$ , by Property 1  $f(s_w, P^*) \leq f(w, P^*) = m - 1 < m$ . Therefore,  $f(s_w, P^*) < m$ , as desired.

To show  $f(s_v, P^*) < m$ , note that by supposition  $r_w - 1 > r_v$ , and from this we immediately obtain  $f(w, P) - 1 > f(v, P)$  by the definition of failure number. Now consider the nodes  $s_v$ , for which

$$f(s_v, P) \leq f(v, P) < f(w, P) - 1 = m - 1 \implies f(s_v, P^*) - 1 < m - 1,$$

Where the first inequality is an application of Property 1, and the implication follows by substitution. Therefore  $f(s_v, P^*) < m$  as desired.

Therefore, among all nodes in  $P^*$  whose failure numbers change as a result of the swap, no node has failure number  $m$ , so  $|S^+| = 0$  as claimed. Moreover, since  $f(s, P^*) < m$  for any node  $s$  whose failure number changes as a result of the swap, we also have proven that  $p_k = p_k^*$  for all  $k$  where  $\rho \geq k > m$ . This completes the proof.  $\square$

## 4 An $O(n\rho)$ Algorithm

Our algorithm considers only placements which are balanced. To place  $\rho$  replicas, we start by placing  $\rho$  replicas at the root of the tree, and then proceed to assign these replicas to children of the root. We then recursively carry out the same procedure on each of the children.

Before the recursive procedure begins, we obtain values of  $\ell_i$  at each node by running breadth-first search as a preprocessing phase. The recursive procedure is then executed in two consecutive phases. During the *divide* phase, the algorithm is tasked with allocating  $r(u)$  replicas placed on node  $u$  to the children of  $u$ . After the divide phase, some child nodes are filled, while others remain unfilled. To achieve balance, each unfilled child  $c_i$  will have either  $r(c_i)$  or  $r(c_i) - 1$  replicas placed upon them. The value of  $r(c_i)$  is computed for each  $c_i$  as part of the divide

phase. The algorithm is then recursively called on each unfilled node to obtain values of optimal solutions for their subtrees. Nodes which are filled require no further processing. The output of this call is a pair of two optimal failure aggregates, one supposing  $r(c_i)$  replicas are placed at  $c_i$ , the other supposing  $r(c_i) - 1$  are placed. Given these failure aggregates obtained from each child, the *conquer* phase then chooses whether to place  $r(c_i)$  or  $r(c_i) - 1$  replicas on each unfilled child so as to achieve a lexicominimum failure aggregate for node  $u$  overall. For ease of exposition, we describe an  $O(n\rho)$  version of our algorithm in this section, and prove it correct. In Section 5 then discuss improvements which can be used to obtain an  $O(n + \rho^2)$  algorithm. Finally, we describe some tree transformations which can be used to obtain an  $O(n + \rho \log \rho)$  algorithm in Section 6.

#### 4.1 Divide Phase

When node  $u$  is first considered, it receives at most two possible values for the number of replicas it could be asked to accommodate. Let these be the values  $r(u)$  and  $r(u) - 1$ . Let  $u$  have a list of children indexed  $1, 2, \dots, m$ , with leaf capacities  $\ell_i$  where  $1 \leq i \leq m$ . The divide phase determines which children will be filled and which will be unfilled. Filled children will have  $\ell_i$  replicas placed on them in the optimal solution, while the number of replicas on the unfilled children is determined during the conquer phase.

The set of unfilled children can be determined (without sorting) in an iterative manner using an  $O(m)$  time algorithm similar to that for the Fractional Knapsack problem. The main idea of the algorithm is as follows: in each iteration, at least one-half of the children whose status is currently unknown are assigned a filled/unfilled status. To determine which half, the median capacity child (with capacity  $\ell_{med}$ ) is found using the selection algorithm. Based upon the number of replicas that have not been assigned to the filled nodes, either a) the set of children  $c_i$  with  $\ell_i \geq \ell_{med}$  are labeled as “unfilled” or b) the set of children  $c_i$  with  $\ell_i \leq \ell_{med}$  are labeled as “filled”. The algorithm recurses on the remaining unlabeled children. Pseudocode for this algorithm can be found in Algorithm 1

We briefly sketch the correctness of Algorithm 1. The following invariant holds after every execution of the while loop:

$$\max(F) \cdot (|U| + |M|) < r - \sum_{c_i \in F} \ell_i \leq \min(U) \cdot |U| + \sum_{c_i \in M} \ell_i.$$

When  $U = \emptyset$  or  $F = \emptyset$  the invariant is not well-defined. These conditions are easy to test for:  $U = \emptyset$  if and only if  $\sum \ell_i = r(u)$ , and  $F = \emptyset$  if and only if  $\ell_i > \lfloor \frac{r(u)}{|M|} \rfloor$  for all  $i$ . Hence in what follows, we will work only with cases where  $U \neq \emptyset$  and  $F \neq \emptyset$ . At the end of the algorithm,  $M = \emptyset$ , and the invariant reduces to the following

$$\max(F) < \frac{r - \sum_{c_i \in F} \ell_i}{|U|} \leq \min(U). \quad (1)$$

---

**Algorithm 1:** Determines filled and unfilled nodes

---

```

1 Function Get-Filled( $M, r$ )begin
2    $F \leftarrow \emptyset; U \leftarrow \emptyset;$     //  $F :=$  filled children  $U :=$  unfilled children
3   while  $M \neq \emptyset$  do
4      $\ell_{med} \leftarrow$  median capacity of children in  $M$  ;
5      $M_1 \leftarrow \{c_i \in M \mid \ell_i < \ell_{med}\}$  ;
6      $M_2 \leftarrow \{c_i \in M \mid \ell_i = \ell_{med}\}$  ;
7      $M_3 \leftarrow \{c_i \in M \mid \ell_i > \ell_{med}\}$  ;
8      $x \leftarrow r - \sum_{c_i \in F \cup M_1 \cup M_2} \ell_i$  ;    //  $x$  to be distributed among  $M_3 \cup U$ 
9     if  $x \geq \ell_{med} \cdot (|U| + |M_3|)$  then          //  $M_1 \cup M_2$  guaranteed filled
10       $F \leftarrow F \cup M_1 \cup M_2$  ;
11       $M \leftarrow M - (M_1 \cup M_2)$  ;
12    else                                           //  $M_2 \cup M_3$  guaranteed unfilled
13       $U \leftarrow F \cup M_2 \cup M_3$  ;
14       $M \leftarrow M - (M_2 \cup M_3)$  ;
15  return ( $F, U$ ) ;    // return filled and unfilled children

```

---

Equation 1 indicates that the average number of replicas placed on the unfilled nodes lies between the maximum value of  $F$  and the minimum value of  $U$ . From this, it is easy to see that the labeling is correct. Suppose that some filled child  $c_i \in F$  has been incorrectly classified. This child contains at most  $\ell_i - 1$  replicas, and yet is still unfilled. Moreover, to attain the average, some unfilled child must be assigned at least  $\lceil \frac{r - \sum_{c_i \in F} \ell_i}{|U|} \rceil$  replicas. Taking the difference of the number of replicas assigned to these two unfilled nodes, we have

$$\begin{aligned}
& \left\lceil \frac{r - \sum_{c_i \in F} \ell_i}{|U|} \right\rceil - \ell_i + 1 \\
& > \left\lceil \frac{r - \sum_{c_i \in F} \ell_i}{|U|} \right\rceil - \max(F) + 1 \\
& \geq \left\lceil \frac{r - \sum_{c_i \in F} \ell_i}{|U|} \right\rceil - \max(F) + 2 \geq 2
\end{aligned}$$

which is a violation of the balanced placement property. Therefore, all replicas are correctly classified. This completes the proof sketch.

Suppose we know that we only need to find placements of size  $r(u)$  and  $r(u) - 1$  for node  $u$ . Moreover, we know that in an optimal placement of size  $r(u)$ , each child  $c_i$  only needs to accomodate either  $r(c_i)$  or  $r(c_i) - 1$  replicas. Suppose that optimal placements of size  $r(c_i)$  and  $r(c_i) - 1$  are available at each child  $c_i$ . Theorem 3 shows that these placements are all that is required to compute optimal placements of size  $r(u)$  and also of size  $r(u) - 1$ .

**Theorem 3.** *In any case where  $r(u)$  or  $r(u)-1$  replicas must be balanced among  $k$  unfilled children, it suffices to consider placing either  $\lceil \frac{r(u)-L}{k} \rceil$  or  $\lfloor \frac{r(u)-L-1}{k} \rfloor$  replicas at each unfilled child.*

*Proof.* Let  $s := r(u) - L$ . Suppose  $s \bmod k = 0$ . If  $s$  replicas are placed at  $u$ , then all unfilled children receive exactly  $\frac{s}{k}$  ( $= \lceil \frac{s}{k} \rceil$ ) replicas. If  $s-1$  replicas are placed at  $u$ , one child gets  $\frac{s}{k} - 1 = \lfloor \frac{s-1}{k} \rfloor$  replicas. If instead  $s \bmod k > 0$ , then the average number of replicas on each unfilled child is  $\frac{s}{k} \notin \mathbb{Z}$ . To attain this average using integer values, values both above and below  $\frac{s}{k}$  are needed. However, since the unfilled children must be balanced, whatever values selected must have absolute difference at most 1. The only two integer values satisfying these requirements are  $\lceil \frac{s}{k} \rceil$  and  $\lfloor \frac{s}{k} \rfloor$ . But  $\lfloor \frac{s}{k} \rfloor = \lfloor \frac{s-1}{k} \rfloor$  when  $s \bmod k > 0$ .  $\square$

## 4.2 Conquer Phase

Once the recursive call completes, we combine the results from each of the children to achieve the lexicographic minimum overall. Our task in this phase is to select  $(r(u) - L) \bmod k$  unfilled children on which  $\lceil \frac{r(u)-L}{k} \rceil$  replicas will be placed, and place  $\lfloor \frac{r(u)-L-1}{k} \rfloor$  replicas on the remaining unfilled children. We need to do this in such a way that the resulting placement is lexicominimum. Recall also that we must return two values, one for  $r(u)$  and another for  $r(u) - 1$ . We show how to obtain a solution in the  $r(u) - 1$  case using a greedy algorithm. A solution for  $r(u)$  can easily be obtained thereafter. In this section, when two vectors are compared or summed, we are implicitly making use of an  $O(\rho)$  function for comparing two vectors of length  $\rho$  in lexicographic order.

Let  $\mathbf{a}_i$  (respectively  $\mathbf{b}_i$ ) represent the lexicominimum value of  $\mathbf{f}(P)$  where  $P$  is any placement of  $\lfloor \frac{r(u)-L-1}{k} \rfloor$  (respectively  $\lceil \frac{r(u)-L}{k} \rceil$ ) replicas on child  $i$ . Recall that  $\mathbf{a}_i, \mathbf{b}_i \in \mathbb{N}^{\rho+1}$ , and are available as the result of the recursive call. We solve the optimization problem by encoding the decision to take  $\mathbf{b}_i$  over  $\mathbf{a}_i$  as a decision variable  $x_i \in \{0, 1\}$ , for which either  $x_i = 0$  if  $\mathbf{a}_i$  is selected, or  $x_i = 1$  if  $\mathbf{b}_i$  is selected. The problem can then be described as an assignment of values to  $x_i$  according to the following system of constraints, in which all arithmetic operations are performed point-wise.

$$\min \sum_i \mathbf{a}_i + (\mathbf{b}_i - \mathbf{a}_i)x_i, \quad \text{subj. to: } \sum_i x_i = (r(u) - L) \bmod k. \quad (2)$$

An assignment of  $x_i$  which satisfies the requirements in (2) can be found by computing  $\mathbf{b}_i - \mathbf{a}_i$  for all  $i$ , and greedily assigning  $x_i = 1$  to those  $i$  which have the  $(r(u) - L) \bmod k$  smallest values of  $\mathbf{b}_i - \mathbf{a}_i$ . This is formally stated as

**Theorem 4.** *Let  $\pi := (\pi_1, \pi_2, \dots, \pi_k)$  be a permutation of  $\{1, 2, \dots, k\}$  such that:*

$$\mathbf{b}_{\pi_1} - \mathbf{a}_{\pi_1} \leq_L \mathbf{b}_{\pi_2} - \mathbf{a}_{\pi_2} \leq_L \dots \leq_L \mathbf{b}_{\pi_k} - \mathbf{a}_{\pi_k}.$$

*If vector  $\mathbf{x} = \langle x_1, \dots, x_k \rangle$  is defined according to the following rules: set  $x_{\pi_i} = 1$  iff  $i < (r(u) - L) \bmod k$ , else  $x_{\pi_i} = 0$ , then  $\mathbf{x}$  is an optimal solution to (2).*

The following Lemma greatly simplifies the proof of Theorem 4.

**Lemma 3.**  $\langle \mathbb{Z}^n, + \rangle$  forms a linearly-ordered group under  $\leq_L$ . In particular, for any  $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{Z}^n$ ,  $\mathbf{x} \leq_L \mathbf{y} \implies \mathbf{x} + \mathbf{z} \leq_L \mathbf{y} + \mathbf{z}$ .

A straight-forward proof of Lemma 3 can be found in the appendix.

*Proof (Proof of Theorem 4).* First, notice that a solution to (2) which minimizes the quantity  $\sum_i (\mathbf{b}_i - \mathbf{a}_i)x_i$  also minimizes the quantity  $\sum_i \mathbf{a}_i + (\mathbf{b}_i - \mathbf{a}_i)x_i$ . It suffices to minimize the former quantity, which can be done by considering only those values of  $(\mathbf{b}_i - \mathbf{a}_i)$  for which  $x_i = 1$ . For convenience, we consider  $\mathbf{x}$  to be the characteristic vector of a set  $S \subseteq \{1, \dots, k\}$ . We show that no other set  $S'$  can yield a characteristic vector  $\mathbf{x}'$  which is strictly better than  $\mathbf{x}$  as follows.

Let  $\alpha := (r(u) - L) \bmod k$ , and let  $S' := \{\pi_1, \dots, \pi_{\alpha-1}\}$  be the first  $\alpha - 1$  entries of  $\pi$  taken as a set. Suppose that there is some  $S'$  which represents a feasible assignment of variables to  $\mathbf{x}'$  for which  $\mathbf{x}'$  is a strictly better solution than  $\mathbf{x}$ .  $S' \subseteq \{1, \dots, k\}$ , such that  $|S'| = \alpha - 1$ , and  $S' \neq S$ . Since  $S' \neq S$ , and  $|S'| = |S|$  we have that  $S - S' \neq \emptyset$  and  $S' - S \neq \emptyset$ . Let  $i \in S - S'$  and  $j \in S' - S$ . We claim that we can form a better placement,  $S^* = (S' - \{j\}) \cup \{i\}$ . Specifically,

$$\sum_{\ell \in S^*} (\mathbf{b}_\ell - \mathbf{a}_\ell) \leq_L \sum_{m \in S'} (\mathbf{b}_m - \mathbf{a}_m) . \quad (3)$$

which implies that replacing a single element in  $S'$  with one from  $S$  does not cause the quantity minimized in (2) to increase.

To prove (3) note that  $j \notin S$  and  $i \in S \implies (\mathbf{b}_i - \mathbf{a}_i) \leq_L (\mathbf{b}_j - \mathbf{a}_j)$ . We now apply Lemma 3, setting  $\mathbf{x} = (\mathbf{b}_i - \mathbf{a}_i)$ ,  $\mathbf{y} = (\mathbf{b}_j - \mathbf{a}_j)$ , and  $\mathbf{z} = \sum_{\ell \in (S^* - \{i\})} (\mathbf{b}_\ell - \mathbf{a}_\ell)$ . This yields

$$\sum_{\ell \in (S^* - \{i\})} (\mathbf{b}_\ell - \mathbf{a}_\ell) + (\mathbf{b}_i - \mathbf{a}_i) \leq_L \sum_{\ell \in (S^* - \{i\})} (\mathbf{b}_\ell - \mathbf{a}_\ell) + (\mathbf{b}_j - \mathbf{a}_j) .$$

But since  $S^* - \{i\} = S' - \{j\}$ , we have that

$$\sum_{\ell \in (S^* - \{i\})} (\mathbf{b}_\ell - \mathbf{a}_\ell) + (\mathbf{b}_i - \mathbf{a}_i) \leq_L \sum_{m \in (S' - \{j\})} (\mathbf{b}_m - \mathbf{a}_m) + (\mathbf{b}_j - \mathbf{a}_j) . \quad (4)$$

Clearly, (4)  $\implies$  (3), thereby proving (3). This shows that any solution which is not  $S$  can be modified to swap in one extra member of  $S$  without increasing the quantity minimized in (2). By induction, it is possible to include every element from  $S$ , until  $S$  itself is reached. Therefore,  $\mathbf{x}$  is an optimal solution to (2).  $\square$

In the algorithm, we find an optimal solution to (2) by assigning  $\lceil \frac{r(u)-L-1}{k} \rceil$  replicas to those children where  $i$  is such that  $1 \leq i < (r(u) - L) \bmod k$ , and  $\lfloor \frac{r(u)-L}{k} \rfloor$  replicas to those remaining. To do this, we find the unfilled child having the  $((r(u) - L) \bmod k)^{th}$  largest value of  $\mathbf{b}_i - \mathbf{a}_i$  using linear-time selection, and use the partition procedure from quicksort to find those children having values below the selected child. This takes time  $O(k\rho)$  at each node.

At the end of the conquer phase, we compute and return the sum<sup>1</sup>

$$\sum_{i < (r(u)-L) \bmod k} \mathbf{b}_i + \sum_{i \geq (r(u)-L) \bmod k} \mathbf{a}_i + \sum_{j: \text{filled}} \mathbf{f}(P_j) + \mathbf{1}_{r(u)-1}, \quad (5)$$

where  $P_j$  is the placement of  $\ell_j$  replicas on child  $j$  and  $\mathbf{1}_{r(u)-1}$  is a vector of length  $\rho$  having a one in entry  $r(u) - 1$  and zeroes everywhere else. The term  $\mathbf{1}_{r(u)-1}$  accounts for the failure number of  $u$ . This sum gives the value of an optimal placement of size  $r(u) - 1$ . Note there are  $k + 1$  terms in the sum, each of which is a vector of length at most  $\rho + 1$ . Both computing the sum and performing the selection take  $O(k\rho)$  time at each node, yielding  $O(n\rho)$  time overall.

We have only focused upon computing the *value* of the optimal solution. The solution itself can be recovered easily by storing the decisions made during the conquer phase at each node, and then combining them to output an optimal placement.

## 5 An $O(n + \rho^2)$ Algorithm

An  $O(n + \rho^2)$  running time can be achieved by an  $O(n)$  divide phase, and an  $O(\rho^2)$  conquer phase. The divide phase already takes at most  $O(n)$  time overall, so to achieve our goal, we concern ourselves with optimizing the conquer phase. The conquer phase can be improved upon by making two changes. First, we modify the vector representation used for return values. Second, we transform the structure of the tree to avoid pathological cases.

In the remainder of the paper, we will use array notation to refer to entries of vectors. For a vector  $\mathbf{v}$ , the  $k^{\text{th}}$  entry of  $\mathbf{v}$  is denoted  $\mathbf{v}[k]$ .

**Compact Vector Representation** Observe that the maximum failure number returned from child  $c_i$  is  $r(c_i)$ . This along with Property 1 implies that the vector returned from  $c_i$  will have a zero in indices  $\rho, \rho - 1, \dots, r(c_i) + 1$ . To avoid wasting space, we modify the algorithm to return vectors of length only  $r(c_i)$ . At each node, we then compute (5) by summing entries in increasing order of their index. Specifically, to compute  $\mathbf{v}_1 + \mathbf{v}_2 + \dots + \mathbf{v}_k$ , where each vector  $\mathbf{v}_j$  has length  $r(c_i)$ , we first allocate an empty vector  $\mathbf{w}$ , of size  $r(c_i)$ , to store the result of the sum. Then, for each vector  $\mathbf{v}_j$ , we set  $\mathbf{w}[i] \leftarrow \mathbf{w}[i] + \mathbf{v}_j[i]$  for indices  $i$  from 0 up to  $r(c_i)$ . After all vectors have been processed,  $\mathbf{w} = \mathbf{v}_1 + \dots + \mathbf{v}_k$ . This algorithm takes  $r(c_1) + \dots + r(c_k) = O(r(u))$  time. Using smaller vectors also implies that the  $((r(u) - L) \bmod k)^{\text{th}}$  best child is found in  $O(r(u))$  time, since each unfilled child returns a vector of size at most  $O(\frac{r(u)}{k})$ , and there are only  $k$  unfilled children to compare. With these modifications the conquer phase takes  $O(r(u))$  time at node  $u$ .

<sup>1</sup> In the mentioned sum we assume for notational convenience, that the vectors have been indexed in increasing order of  $\mathbf{b}_i - \mathbf{a}_i$ , although the algorithm performs no such sorting.

**Tree Transformations** Note that for each  $i$ , nodes at depth  $i$  have  $O(\rho)$  replicas placed on them in total. We can therefore achieve an  $O(\rho^2)$  time conquer phase overall by ensuring that the conquer phase only needs to occur in at most  $O(\rho)$  levels of the tree. To do this, we observe that when  $r(u) = 1$ , any leaf with minimum depth forms an optimal placement. Recursive calls can therefore be stopped once  $r(u) = 1$ . To ensure that  $r(u) = 1$  after  $O(\rho)$  levels, we contract paths on which all nodes have degree two into a single pseudonode during the preprocessing phase. The length of this contracted path is stored in the pseudonode, and is accounted for when computing the sum. This suffices to ensure  $r(u)$  decreases by at least one at each level, yielding an  $O(n + \rho^2)$  algorithm.

## 6 An $O(n + \rho \log \rho)$ Algorithm

In this section, we extend ideas about tree transformation from the last section to develop an algorithm in which the conquer phase only needs to occur in at most  $O(\log \rho)$  levels. We achieve this by refining the tree transformations described in Section 5.

To ensure that there are only  $O(\log \rho)$  levels in the tree, we transform the tree so as to guarantee that as the conquer phase proceeds down the tree,  $r(u)$  decreases by at least a factor of two at each level. This happens automatically when there are two or more unfilled nodes at each node, since to balance the unfilled children, at most  $\lceil \frac{r(u)-L}{2} \rceil$  replicas will be placed on each of them. Problems can therefore only arise when a tree has a path of nodes each of which have a single, unfilled child. We call such a path a *degenerate chain*. By detecting and contracting all such degenerate chains, we can achieve an  $O(\rho \log \rho)$  conquer phase.

Fig. 7(a) illustrates a degenerate chain. In this figure, each  $T_i$  with  $1 \leq i \leq t-1$  is the set of all descendant nodes of  $v_i$  which are filled. Thus,  $v_1, \dots, v_{t-1}$  each have only a single unfilled child (since each  $v_i$  has  $v_{i+1}$  as a child). In contrast, node  $v_t$  has at least two unfilled children. It is easy to see that if the number of leaves in each  $T_i$  is  $O(1)$  then  $t$ , the length of the chain, can be as large as  $O(\rho)$ . This would imply that there can be  $O(\rho)$  levels in the tree where the entire conquer phase is required. To remove degenerate chains, we contract nodes  $v_1, \dots, v_{t-1}$  into a single pseudonode  $w$ , as in Fig. 7(b). However, we must take care to ensure that the pair of vectors which pseudonode  $w$  returns takes into account contributions from the entire contracted structure. We will continue to use  $v_i$  and  $T_i$  throughout the remainder of this section to refer to nodes in a degenerate chain.

To find and contract degenerate chains, we add an additional phase, the *transform* phase, which takes place between the divide and conquer phases. Recall that after the divide phase, the set of filled and unfilled children are available at each node. Finding nodes in a degenerate chain is therefore easily done via a breadth-first search. We next consider what information must be stored in the pseudonode, to ensure that correct results are maintained.



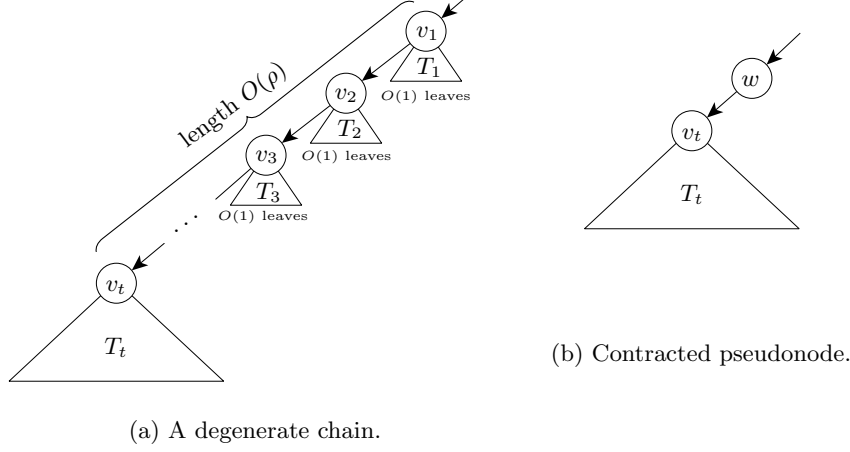


Fig. 7: Illustration of a degenerate chain in which each  $v_i$  where  $1 \leq i \leq t-1$  represents a node which has a single unfilled child. All filled descendants of node  $v_i$  are collectively represented as  $T_i$ . In the figure on the right, nodes  $v_1, \dots, v_{t-1}$  have been contracted into pseudonode  $w$ .

Let  $(\mathbf{a}_w, \mathbf{b}_w)$  be the pair of values which will be returned by pseudonode  $w$  at the end of the conquer phase. In order for the transformation to be correct, the vectors  $(\mathbf{a}_w, \mathbf{b}_w)$  must be the same as those which would have been returned at node  $v_1$  had no transformation occurred. To ensure this, we must consider and include the contribution of each node in the set  $T_1 \cup \dots \cup T_{t-1} \cup \{v_1, \dots, v_{t-1}\}$ . It is easy to see that the failure numbers of nodes in  $\{v_1, \dots, v_{t-1}\}$  depend only upon whether  $r(v_t)$  or  $r(v_t) - 1$  replicas are placed on node  $v_t$ , while the filled nodes in sets  $T_1, \dots, T_{t-1}$  have no such dependency. Observe that if  $r(v_t)$  replicas are placed on  $v_t$ , then  $r(v_i)$  replicas are placed at each node  $v_i$ . If instead  $r(v_t) - 1$  replicas are placed, then  $r(v_i) - 1$  replicas are placed at each  $v_i$ . Since values of  $r(v_i)$  are available at each node after the divide phase, enough information is present to contract the degenerate chain before the conquer phase is performed.

The remainder of this section focuses on the technical details needed to support our claim that the transform phase can be implemented in time  $O(n + \rho \log \rho)$  overall. Let  $S_w := T_1 \cup \dots \cup T_{t-1} \cup \{v_1, \dots, v_{t-1}\}$ , and let the contribution of nodes in  $S_w$  to  $\mathbf{a}_w$  and  $\mathbf{b}_w$  be given by vectors  $\mathbf{a}$  and  $\mathbf{b}$  respectively. The transform phase is then tasked with computing  $\mathbf{a}$  and  $\mathbf{b}$ , and contracting the degenerate chain. We will show that this can be done in time  $O(|S_w| + r(v_1))$  for each pseudonode  $w$ .

Pseudocode for the transform phase is given in Algorithm 2. The transform phase is started at the root of the tree by invoking  $\text{Transform}(\text{root}, \text{false}, \rho)$ . **Transform** is a modified recursive breadth-first search. As the recursion proceeds down the tree, each node is tested to see if it is part of a degenerate chain (lines

---

**Algorithm 2:** Transform phase

---

```

1 Function Transform( $u, chain, r(v_1)$ ) begin
2   if  $u$  has two or more unfilled children then
3     foreach child  $c_i$  unfilled do
4        $(-, -, -, x) \leftarrow \text{Transform}(c_i, false, \perp)$  ;
5        $c_i \leftarrow x$  ;
6     if  $chain = false$  then return  $(\perp, \perp, \perp, u)$  ;
7     else return  $(\mathbf{0}_{r(v_1)+1}, \mathbf{0}_{r(v_1)+1}, \mathbf{0}_{r(v_1)+1}, u)$  ;      //  $3 \cdot O(r(v_1))$  time
8   if  $u$  has one unfilled child,  $v$  then
9     if  $chain = false$  then
10      // pass  $r(v)$  as max vector length
11       $(\mathbf{a}, \mathbf{b}, \mathbf{f}, x) \leftarrow \text{Transform}(v, true, r(v))$  ;
12    else
13       $(\mathbf{a}, \mathbf{b}, \mathbf{f}, x) \leftarrow \text{Transform}(v, true, r(v_1))$  ;
14    foreach filled child  $c_i$  do
15       $\text{Filled}(c_i, \mathbf{f})$  ;                                //  $O(n_i)$  time
16     $k \leftarrow \sum_i \ell_i + r(v) - 1$  ;
17     $\mathbf{a}[k+1] \leftarrow \mathbf{a}[k+1] + 1$  ;
18     $\mathbf{b}[k] \leftarrow \mathbf{b}[k] + 1$  ;
19    if  $chain = false$  then
20       $x \leftarrow \text{Make-Pseudonode}(\mathbf{a}, \mathbf{b}, \mathbf{f}, x)$ 
21    return  $(\mathbf{a}, \mathbf{b}, \mathbf{f}, x)$ 

```

---

2 and 8). If a node is not part of a degenerate chain, the call continues on all unfilled children (line 3). The first node ( $v_1$ ) in a degenerate chain is marked by passing down  $chain \leftarrow true$  at lines 11 and 13. The value of  $r(v_1)$  is also passed down to the bottom of the chain at lines 11 and 13. Once the bottom of the chain (node  $v_t$ ) has been reached, the algorithm allocates memory for three vectors,  $\mathbf{a}$ ,  $\mathbf{b}$  and  $\mathbf{f}$ , each of size  $r(v_1) + 1$  (line 7). These vectors are then passed up through the entire degenerate chain (line 21), along with node  $u$ , whose use will be explained later. When a node  $u$  in a degenerate chain receives  $\mathbf{a}$ ,  $\mathbf{b}$ , and  $\mathbf{f}$ ,  $u$  adds its contribution to each vector (lines 14-18). The contribution of node  $u$  consists of two parts. First, the contribution of the filled nodes is added to  $\mathbf{f}$  by invoking a special **Filled** subroutine (see Algorithm 3) which computes the sum of the failure aggregates of each filled child of  $u$  (lines 14-15). Note that **Filled** uses pass-by-reference semantics when passing in the value of  $\mathbf{f}$ . The contribution of node  $u$  itself is then added, by summing the number of leaves in all of the filled children, and the number of replicas on the single unfilled child,  $v$  (lines 16-18). By the time that the recursion reaches the start of the chain on the way back up (line 19), all nodes have added their contribution, and the pseudonode is created and returned (line 20).

The transformation takes place as **Transform** is returned back up the tree. At the end of the degenerate chain, node  $v_t$  is returned (lines 6-7), and this value is

---

**Algorithm 3:** Computes failure aggregate of filled nodes

---

```

1 Function Filled( $u, \mathbf{f}$ ) begin
2   else if  $u$  is a leaf then
3      $\mathbf{f}[0] \leftarrow \mathbf{f}[0] + 1$ ;
4     return;
5   foreach child  $c_i$  do
6     Filled( $c_i, \mathbf{f}$ )
7    $a \leftarrow \sum_i \ell_i$ ;
8    $\mathbf{f}[a] \leftarrow \mathbf{f}[a] + 1$ ;
9   return;

```

---

passed along the length of the entire chain (line 21), until reaching the beginning of the chain, where the pseudonode is created and returned (line 20). When the beginning of the chain is reached, the parent of  $v_1$  updates its reference (line 5) to refer to the newly created pseudonode. At line 5 note that if  $c_i$  was *not* the beginning of a degenerate chain,  $x = c_i$  and the assignment has no effect (see lines 6-7).

We provide pseudocode for the **Filled** and **Make-Pseudonode** subroutines in Algorithms 3 and 4. The **Make-Pseudonode** subroutine runs in  $O(1)$  time. It is easy to see that the **Filled** routine runs in  $O(n_i)$  time, where  $n_i$  is the number of nodes in the subtree rooted at child  $c_i$ . The **Transform** routine therefore takes  $O(|T_i|)$  time to process a single node  $v_i$ . The time needed for **Transform** to process an entire degenerate chain is therefore  $O(|S_w|) + 3 \cdot O(r(v_1))$ , where the  $3 \cdot O(r(v_1))$  term arises from allocating memory for vectors  $\mathbf{a}$ ,  $\mathbf{b}$  and  $\mathbf{f}$  at the last node of the chain.

When we sum this time over all degenerate chains, we obtain a running time of  $O(n + \rho \log \rho)$  for the transform phase. To reach this result, we examine the sum of  $r(v_1)$  for all pseudonodes at level  $i$ . Since there are at most  $\rho$  replicas at each level  $i$ , this sum can be at most  $O(\rho)$  in any level. There are only  $O(\log \rho)$  levels where  $r(u) > 1$  after degenerate chains have been contracted, thus, pseudonodes can be only be present in the first  $O(\log \rho)$  levels of the tree. Therefore the  $3 \cdot O(r(v_1))$  term sums to  $O(\rho \log \rho)$  overall. Since  $|S_w|$  clearly sums to  $O(n)$  overall, the transform phase takes at most  $O(n + \rho \log \rho)$  time.

Finally, after the transformation has completed, we can ensure that the value of  $r(u)$  decreases by a factor of two at each level. This implies that there are only  $O(\log \rho)$  levels where the conquer phase needs to be run in its entirety. Therefore, the conquer phase takes  $O(\rho \log \rho)$  time overall. When combined with the  $O(n)$  divide phase and the  $O(n + \rho \log \rho)$  transform phase, this yields an  $O(n + \rho \log \rho)$  algorithm for solving replica placement in a tree.

---

**Algorithm 4:** Creates and returns a new pseudonode

---

```
1 Function Make-Pseudonode(a, b, f, x)begin  
2   allocate a new node node;  
3   node.a  $\leftarrow a + f$ ;  
4   node.b  $\leftarrow b + f$ ;  
5   node.child  $\leftarrow x$ ;  
6   return node
```

---

## 7 Conclusion

In this paper, we formulate the replica placement problem and show that it can be solved by a greedy algorithm in  $O(n^2\rho)$  time. In search of a faster algorithm, we prove that any optimal placement in a tree must be balanced. We then exploit this property to give a  $O(n\rho)$  algorithm for finding such an optimal placement. The running time of this algorithm is then improved, yielding an  $O(n + \rho \log \rho)$  algorithm. An interesting next step would consist of proving a lower bound for this problem, and seeing how our algorithm compares. In future work we plan to consider replica placement on additional classes of graphs, such as special cases of bipartite graphs.

We would like to acknowledge insightful comments from S. Venkatesan and Balaji Raghavachari during meetings about results contained in this paper, as well as comments from Conner Davis on a draft version of this paper.

## References

1. Bakaloglu, M., Wylie, J.J., Wang, C., et. al: On correlated failures in survivable storage systems. Tech. Rep. CMU-CS- 02-129, Carnegie Mellon University (2002)
2. Blume, L., Easley, D., Kleinberg, J., Kleinberg, R., Tardos, E.: Which networks are least susceptible to cascading failures? In: Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on. pp. 393–402 (Oct 2011)
3. Chen, M., Chen, W., Liu, L., et. al: An analytical framework and its applications for studying brick storage reliability. In: Proceedings of 26th International Symposium on Reliable Distributed Systems. pp. 242–252. IEEE Computer Society (2007)
4. Ford, D., Labelle, F., Popovici, F.I., Stokely, M., Truong, V.A., Barroso, L., Grimes, C., Quinlan, S.: Availability in globally distributed storage systems. In: Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation. USENIX, Berkeley, CA (2010)
5. Hu, X.D., Jia, X.H., Du, D.Z., Li, D.Y., Huang, H.J.: Placement of data replicas for optimal data availability in ring networks. Journal of Parallel and Distributed Computing 61(10), pp. 1412 – 1424 (2001)
6. Kim, J., Dobson, I.: Approximating a loading-dependent cascading failure model with a branching process. IEEE Transactions on Reliability 59, pp. 691–699 (Dec 2010)
7. Lian, Q., Chen, W., Zhang, Z.: On the impact of replica placement to the reliability of distributed brick storage systems. In: 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05). pp. pp. 187–196. IEEE (2005)

8. Nath, S., Yu, H., Gibbons, P.B., Seshan, S.: Subtleties in tolerating correlated failures in wide-area storage systems. In: 3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings. pp. 225–238 (2006)
9. Nie, L., Liu, J., Zhang, H., Xu, Z.: On the inapproximability of minimizing cascading failures under the deterministic threshold model. *Information Processing Letters* 114, pp. 1–4 (2014)
10. Pezoa, J., Hayat, M.: Reliability of heterogeneous distributed computing systems in the presence of correlated failures. *IEEE Transactions on Parallel and Distributed Systems*, 25(4), pp. 1034–1043 (April 2014)
11. Pletz, J.: The price of failure: Data-center power outage cost sears \$2.2m in profit. <http://www.chicagobusiness.com/article/20130604/BLOGS11/130609948/the-price-of-failure-data-center-power-outage-cost-sears-2-2m-in-profit> (Jun 2013)
12. Shekhar, S., Wu, W.: Optimal placement of data replicas in distributed database with majority voting protocol. *Theoretical Computer Science* 258, pp. 555 – 571 (2001)
13. Verge, J.: How a switch failure in utah took out four big hosting providers. <http://www.datacenterknowledge.com/archives/2013/08/05/how-did-the-failure-of-network-switches-at-a-little-known-data-center-in-provo-utah-knock-four-major-services-and-millions-of-web-pages-offline> (Sep 2013)
14. Weatherspoon, H., Moscovitz, T., Kubiatowicz, J.: Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In: *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, 2002. pp. 362–367 (2002)
15. Zhang, Z., Wu, W., Shekhar, S.: Optimal placements of replicas in a ring network with majority voting protocol. *J. Parallel Distrib. Comput.* 69(5), 461–469 (May 2009)
16. Zhu, Y., Yan, J., Sun, Y., He, H.: Revealing cascading failure vulnerability in power grids using risk-graph. *IEEE Transactions on Parallel and Distributed Systems*, 25(12), pp. 3274–3284 (Dec 2014)

## Appendix

### Proof of Property 1

The following property from Section 2 is an easy result regarding failure numbers which is used in the proofs of Theorems 1 and 2.

*Proof (Proof of Property 1).* Suppose that in  $P$  there are  $k_i$  replicas placed on leaves in the subtree rooted at  $i$ . If  $i$  fails,  $k_i$  replicas fail, yielding  $s(i, P) = \rho - k_i$ . Let  $k_j$  be the number of replicas in the subtree rooted at  $j$ . Clearly,  $k_j \leq k_i$ , yielding the result.  $\square$

### NP-Hardness of Problem 1

However, Problem 1 is NP-hard to solve exactly or approximately. In particular, we can reduce the well-known problems of INDEPENDENT SET (IS) and DOMINATING SET (DS) to Problem 1. The reduction from DS shows that minimizing only the first entry of  $f(P)$ ,  $f_r$ , is NP-hard, while the reduction from

IS shows that lexico-minimizing the vector *down to* the second-to-last entry,  $f_1$  is NP-hard. That is to say, if it were possible to lexico-minimize the vector  $\langle f_r, \dots, f_2, f_1 \rangle$  in polynomial time, it would imply  $P = NP$ .

### Proof of Lemma 3

The proof of Theorem 4 is greatly simplified through use of an algebraic property of addition on  $\mathbb{Z}^n$  under lexicographic order. Recall that a *group* is a pair  $\langle S, \cdot \rangle$ , where  $S$  is a set, and  $\cdot$  is a binary operation which is 1) closed for  $S$ , 2) is associative, and has both 3) an identity and 4) inverses. A *linearly-ordered group* is a group  $G = \langle S, \cdot \rangle$ , along with a linear-order  $\leq$  on  $S$  in which for all  $x, y, z \in S$ ,  $x \leq y \implies x \cdot z \leq y \cdot z$ , i.e. the linear-order on  $G$  is *translation-invariant*. Lemma 3 states that  $\mathbb{Z}^n$  under  $\leq_L$  has such a property.

*Proof (Proof of Lemma 3).* It is well-known that  $G = \langle \mathbb{Z}^n, + \rangle$  is a group. To show  $G$  is linearly-ordered, it suffices to show that

$$\forall \mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{Z}^n : \mathbf{x} \leq_L \mathbf{y} \implies \mathbf{x} + \mathbf{z} \leq_L \mathbf{y} + \mathbf{z} .$$

If  $\mathbf{x} = \mathbf{y}$  then surely  $\mathbf{x} + \mathbf{z} = \mathbf{y} + \mathbf{z} \implies \mathbf{x} + \mathbf{z} \leq_L \mathbf{y} + \mathbf{z}$ .

If instead,  $\mathbf{x} \leq_L \mathbf{y}$ , then let  $k = \min_{\mathbf{x}_i < \mathbf{y}_i} i$ . Note that for all  $i$  with  $1 \leq i < k$ ,  $\mathbf{x}_i = \mathbf{y}_i$ , and that  $\mathbf{x}_k < \mathbf{y}_k$ . Consider  $\mathbf{x} + \mathbf{z}$  and  $\mathbf{y} + \mathbf{z}$ . Surely, for all  $1 \leq i < k$ ,  $(\mathbf{x} + \mathbf{z})_i = (\mathbf{y} + \mathbf{z})_i$ , since  $\mathbf{x}_i = \mathbf{y}_i \implies \mathbf{x}_i + \mathbf{z}_i = \mathbf{y}_i + \mathbf{z}_i$ . Likewise,  $(\mathbf{x} + \mathbf{z})_k < (\mathbf{y} + \mathbf{z})_k$ , since  $\mathbf{x}_k < \mathbf{y}_k \implies \mathbf{x}_k + \mathbf{z}_k < \mathbf{y}_k + \mathbf{z}_k$ .

Therefore,  $\mathbf{x} \leq_L \mathbf{y} \implies \mathbf{x} + \mathbf{z} \leq_L \mathbf{y} + \mathbf{z}$ . □